



# Device Driver Programming

What's **not** in the Books



# Table of Contents

- Targeted Audience
- General considerations
- Driver Identification
- Driver Interfaces



# Audience

- Not targeted at beginners in device driver programming
- At least basic skills and experience are assumed
- No discussion of basic device driver operation
- Covers OS/2 protected mode drivers only, OS/2 virtual DOS machines are not addressed



# Paradigma

- OS/2 is used in highly dynamic environments now:
  - »Plug and Play« buses like
    - PCCard/Cardbus sockets
    - USB
    - Device bays with hot-swap features
  - Machines with power management
- Device drivers need to handle such hardware
- Programmers have to care about that



# Device Driver Programming

## Driver Identification



# Driver Identification

- Device drivers shall identify themselves verbosely wherever this is possible:
  - in the device driver file (BldLevel)
  - in the resource manager tree (Hardware Manager)
  - in the PCCard manager (PC Card Director)
- This helps you both in development and support
- It gives users confidence in their system setup





# Identification BldLevel

The information shown by the BldLevel utility is stored in the file description:

```
DESCRIPTION "@#DANI:1.5#@##1## 17.9.2002 12:57:21  
Nachtigall:: ::1B::@@ Adapter Driver for ST506/IDE DASD"
```

The result is then

```
Signature:      @#DANI:1.5#@##1## 17.9.2002 12:57:21  
  Nachtigall::  ::1B::@@ Adapter Driver for ST506/IDE DASD  
Vendor:        DANI  
Revision:      1.05  
Date/Time:     17.9.2002 12:57:21  
Build Machine: Nachtigall  
File Version:  1.5.1  
Description:   Adapter Driver for ST506/IDE DASD
```



# Identification Resource Manager

- Even if your driver doesn't handle any hardware, i.e. is a software-only driver, register it with the resource manager
- If your driver shows up in the resource manager device tree (check with RMView /D) then
  - you know that your initialization code is fine
  - the user knows that the driver is successfully loaded





# Identification Resource Manager

```
UCHAR DrvrNameTxt[]  = "DANIS506.ADD";
UCHAR DrvrDescrTxt[] = "DMA Adapter Driver for ST506/IDE DASD";
UCHAR VendorNameTxt[] = "Dani";
DRIVERSTRUCT DriverStruct = {
    DrvrNameTxt,          /* DriverName          */
    DrvrDescrTxt,         /* DriverDescription   */
    VendorNameTxt,        /* VendorName          */
    CMVERSION_MAJOR,      /* MajorVersion        */
    CMVERSION_MINOR,      /* MinorVersion        */
    YEAR,MONTH,DAY,       /* Date                */
    0,                    /* DriverFlags          */
    DRT_ADDDM,            /* DriverType           */
    DRS_ADD,              /* DriverSubType        */
    NULL                  /* DriverCallback       */
};
```

generates this RMView /D output:

```
Driver: DANIS506.ADD  -  DMA Adapter Driver for ST506/IDE DASD
Vendor: Dani   Version: 1.1   Date (MDY): 9/15/2002
Flag: STATIC   Type-Subtype: ADDDM - ADD
```



# Identification PC Card Director

A PCCard client driver registration like this one

```
struct CI_Info ClientInfo = {
    0,
    sizeof (ClientInfo),
    ATB_IOClient | ATB_Insert4Sharable | ATB_Insert4Exclusive,
    VERSION,
    0,
    ((YEAR - 1980) << 9) | (MONTH << 5) | DAY,
    offsetof (struct CI_Info, CI_Name),
    sizeof (ClientInfo.CI_Name),
    offsetof (struct CI_Info, CI_Vendor),
    sizeof (ClientInfo.CI_Vendor),
    "DaniS506 EIDE Driver",
    "Copyright Daniela Engert 2002, all rights reserved"
};
```

generates this output in PC Card Director

# Identification PC Card Director

☒ Card Services

**Client:**

<Type> I/O client  
<Handle> 07F4  
<Revision> 01.51  
<Year> 2002  
<Month> 9  
<Day> 24  
<Card Services level> 02.00  
<Client name>  
DaniS506 EIDE Driver  
<Vendor string>  
Copyright DaniS506, 2002. All rights reserved.

Help

Page 3

Resources Version

Client



# Device Driver Programming

## Driver Interfaces



# Driver Interfaces

- Drivers are DLL modules which are **not** linked against any other modules (except kernel)
- Drivers connect **dynamically** at runtime to kernel services or other OS/2 subsystems in kernel space
- Drivers must **not** assume that all subsystems are available on a given installation
- Connections are usually created by a registration/callback scheme





# Driver Interfaces

- A driver **registers** to other services by IDC lookup (to other drivers) or device helper functions (to kernel services)
- if a driver offers services to other parts of the system it **exports** an entry point into its code
- if a driver uses services from other parts it **imports** an entry point into foreign code
- each service needs a full **specification**





# Driver Interfaces

The minimum specification of a service requires:

- a registration method  
(IDC, device helper function, known location)
- a fully protyped service entry point  
(linkage, return type, argument types, ...)
- a list of execution contexts it may be called in  
(init time, task time, interrupt time, ...)
- a list of restrictions (if any)  
(duration, register usage, access privileges)
- a description of the functions provided



# Driver Interfaces

Drivers implemented in C often require assembler stub functions which interface to the C language model to handle

- arguments passed in registers visa stack based arguments in C
- fully saved registers at the interface visa partially clobbered registers in C routines
- setup of DS to the driver's own default data segment
- setup of DS to the other driver's data segment



# Driver Interfaces

Example:

```
int EntryPoint (int function, anytype *ptr);
```

Questions:

- is the function name `_EntryPoint`, `ENTRYPOINT` or `@EntryPoint` ?
- is the data segment already set up ?
- is a near or far return required ?
- is the pointer argument near or far ?
- what's the argument order ?
- who cleans up the arguments ?



# Driver Interfaces

Driver entry points must be **fully** prototyped to avoid unexpected or faulty behaviour

wrong:

```
int EntryPoint (int function, anytype *ptr);
```

right:

```
int _far _cdecl _loadadds EntryPoint (int function,  
                                     anytype _far *ptr);
```

The calling convention, decoration, DS setup, and near/far attributes need to be specified to be independent of compiler options or models.



# Driver Interfaces

Standard driver interfaces are

- Device helper entry point
- Strategy 1 entry point
- Interrupt handler entry points
- Timer handler entry point
- IDC entry point
- Context hook entry points





# Driver Interfaces

The following driver interfaces are optional, but I consider them **mandatory**!

- Resource Manager services
- APM notifications
- PC Card/Cardbus services and notifications

You need them to adapt the driver operation to dynamic environments





# Driver Interfaces

These driver interfaces are required for discovery of supported hardware

- Resource Manager services
- OEMHelp services

You need them to search in the OS/2 device database or to enumerate the PCI bus



# Driver Interfaces

The following driver interfaces are required by particular classes of device drivers only

- Strategy 2 entry points (ADD/FLT/DMD)
- Strategy 3 entry points (DMD)
- NDIS2 entry points (MAC drivers)
- USB entry points (USB drivers)
- MM Stream entry points (Multimedia drivers)
- others



# Strategy 1

- This entry point is exported to the OS/2 kernel in the device header structure
- Modern device drivers should request InitComplete and Shutdown notifications to attach to or detach from other services properly



# Interrupt Handler

- You **must** handle shared interrupts
- PCI interrupts **may** be unshared
- the execution context of interrupt handlers is **restricted**
- the execution time of interrupt handlers must be **short**
- **defer** as much work as possible to task time handling (f.e. context hooks) - but decide wisely



# Interrupt Handler

```
UCHAR SharingMode;

/* attach to interrupt in given sharing mode */

SharingMode = (IRQ->isShared) ? IRQMODE_SHARED : IRQMODE_UNSHARED;
rc = DevHelp_SetIRQ ((NPFN)IRQ->Handler, IRQ->Level, SharingMode);

/* if attach failed and sharing mode was "shared" try "unshared" */

if (rc && IRQ->isShared) {
    rc = DevHelp_SetIRQ ((NPFN)IRQ->Handler, IRQ->Level, IRQMODE_UNSHARED);

    /* if attach failed another time give up */

    if (!rc) {

        /* adjust actual sharing mode */
        IRQ->isShared = FALSE;

    }
}

/* rc == 0 if attach to interrupt succeeded */
```





# Interrupt Handler

```
/* each IRQ entry point handles a list of instances hooked to this IRQ level */
```

```
USHORT FAR _loadds IRQEntry0() { return (HandleIRQ (IHdr[0].npInst) >> 1); }  
USHORT FAR _loadds IRQEntry1() { return (HandleIRQ (IHdr[1].npInst) >> 1); }  
USHORT FAR _loadds IRQEntry2() { return (HandleIRQ (IHdr[2].npInst) >> 1); }  
USHORT FAR _loadds IRQEntry3() { return (HandleIRQ (IHdr[3].npInst) >> 1); }
```

```
USHORT NEAR HandleIRQ (PTRTYPE_INSTANCEDATA npInst) {  
    USHORT Claimed = 0;  
  
    /* walk list of instances attached to this IRQ */  
    for (; NULL != npInst; npInst = npInst->npIntNext)  
        Claimed |= npInst->IntHandler (npInst);  
  
    return (~Claimed);  
}
```

```
/* As long as the driver isn't prepared to handle interrupts */  
/* from a particular hardware we have to catch them anyway to prevent */  
/* the OS/2 IRQ dispatcher from going mad! */
```

```
USHORT NEAR CatchInterrupt (PTRTYPE_INSTANCEDATA npInst) {  
    if (npInst->CheckIRQ (npInst)) {  
        DevHelp_EOI (npInst->IRQLevel);  
        return (1);  
    }  
    return (0);  
}
```





# Interrupt Handler

```
USHORT NEAR Interrupt (PTRTYPE_INSTANCEDATA npInst){
    USHORT Claimed = 0;
    int    rcCheck;

    /* is the interrupt generated by hardware associated with this instance ? */
    /* if not, bail out early */

    if (!(rcCheck = CheckIRQ (npInst)))
        return (Claimed);

    /* so far, the interrupt is possibly from us */
    /* if we expect an interrupt or the interrupt is definitely from us, */
    /* then handle it */
    /* up to this point, interrupts are still enabled in case of a shared IRQ */
    /* the following code is a section which must not be preempted */

    DISABLE
    if ((npInst->Flags & WAIT_INTERRUPT) || (rcCheck == 1)) {
        npInst->Flags &= ~WAIT_INTERRUPT;

        /* there should be an IRQ timeout timer running */

        if (npInst->IRQTimerHandle) {
            ADD_CancelTimer (npInst->IRQTimerHandle); /* cancel the timer, got IRQ */
            npInst->IRQTimerHandle = 0;
            Claimed = 1;
        } /* else spurious */
    }
```



# Interrupt Handler

```
/* reenable IRQ handling both globally and for this particular IRQ */  
  
ENABLE  
DevHelp_EOI (npInst->IRQLevel);  
  
if (Claimed) {  
    /* the actual handler code is running with interrupts enabled! */  
  
    HandleInterruptForInstance (npInst);  
  
}  
  
/* this is a *requirement* ! */  
/* if we fail to do so, the OS/2 IRQ dispatcher will shut down this */  
/* IRQ line */  
  
Claimed = 1;  
  
} else {  
    ENABLE  
    /* spurious */  
}  
return (Claimed);  
}
```

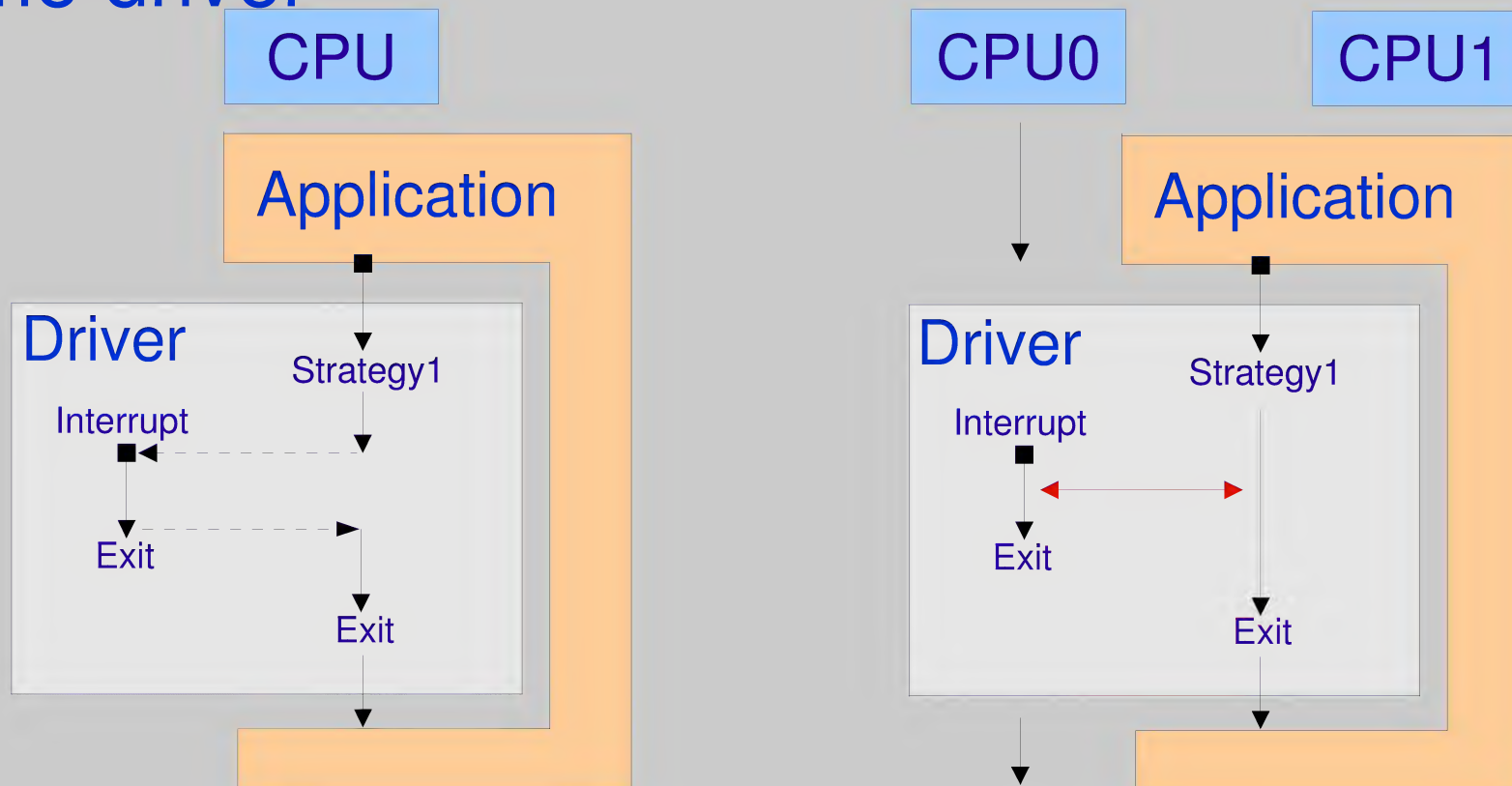


# Interrupt Handler

- Interrupts **preempt** task time execution
- Interrupts **may** preempt interrupt time execution
- Interrupts may preempt their **own** interrupt handlers
- at each preemption, temporary resources (like memory mappings) may become **invalid**
- every preemption **costs** system resources (f.e. stack space)

# Interrupts and SMP

On SMP systems interrupt handlers may execute **concurrently** with other parts of the same driver





# Timer Handlers

- Timer handlers are in fact interrupt handlers, the same rules apply
- if you need multiple timers, the use of ADDCall.lib is recommended. It implements:
  - free running timers
  - one-shot timers
  - each timer may have different time intervals
  - each timer may have arguments





# Timer Handlers

```
/* Expected interrupt timeout routine */
VOID FAR _cdecl IRQTimer (USHORT TimerHandle, ULONG Parameter1, ULONG Parameter2)
{
    /* cancelling the timer makes it an one-shot timer! */
    ADD_CancelTimer (TimerHandle);

    do something useful here
}

/* free running timer with a given call interval */
VOID FAR _cdecl Ticker (USHORT TimerHandle, ULONG Parameter1, ULONG Parameter2)
{
    do something useful here
}

/* Initialize timer pool */

ADD_InitTimer (TimerPool, sizeof (TimerPool));

/* start a timer to call function 'Ticker' with arguments arg1 and arg2 */
/* repeatedly after each TICKER_INTERVAL milliseconds */

ADD_StartTimerMS (&TickerHandle, TICKER_INTERVAL, (PFN)Ticker, arg1, arg2);

/* stop all timer processing and destroy timer pool */

ADD_DeInstallTimer();
```





# IDC Entry Point

- The presence of an IDC entry point and its location within the driver's default code segment is advertised in the device driver header structure
- you need to specify in which execution contexts your IDC services may be called
- if you allow calls in interrupt context the same rules as for interrupt handlers apply
- most likely you need an assembler stub



# IDC Entry Point

```
EXTRN  _IDCHandler:NEAR/FAR
```

```
; VOID NEAR/FAR _cdecl IDCHandler (/* any argument type */)
```

```
; IDC stub to C handler routines
```

```
; needs to be located in the driver default code segment
```

```
PUBLIC _IDCStub
```

```
_IDCStub PROC FAR
```

```
PUSH ES
```

```
PUSH DS
```

```
PUSHAD
```

```
; handle arguments here to match the handler function prototype
```

```
MOV DS, CS:[_DSSel]
```

```
CALL _IDCHandler
```

```
POPAD
```

```
POP DS
```

```
POP ES
```

```
RET
```

```
_IDCStub END
```

```
_DSSel DW SEG _DATA
```



# Context Hooks

- Context hooks do deferred task time processing of interrupt time events (similar to DPCs in WindowsNT)
- Context hooks are executed after all interrupt processing at the next schedule point
- Context hooks can be armed only once until the next execution of the context hook code, multiple invocations need to be queued
- most likely you need an assembler stub



# Context Hooks

```
EXTRN  _CtxHookHandler:NEAR/FAR
```

```
; context hook handler entry points need to be located in the  
; default code segment!
```

```
; VOID NEAR/FAR _cdecl CtxHookHandler (/* any argument type */) 
```

```
PUBLIC _CtxHookStub
```

```
_CtxHookStub PROC FAR
```

```
PUSH ES  
PUSH DS  
PUSHAD
```

```
PUSH EAX ; stack frame is compatible to any data type  
MOV DS, CS:[_DSSel]  
CALL _CtxHookHandler  
ADD SP, 4
```

```
POPAD  
POP DS  
POP ES  
RET
```

```
_CtxHookStub ENDP
```

```
_DSSel DW SEG _DATA
```



# APM Events

- APM event callbacks are possibly called in interrupt context, so the general interrupt handling rules apply
- if necessary, defer APM processing to task time by using a context hook (f.e. device reinitialization after system resume)
- the OS/2 APM subsystem may not be available even if APM is active; the driver needs to handle this scenario gracefully





# APM Registration

```
/* attach to APM at processing of the InitComplete request packet */

{
    UCHAR noAPM;

    /* attach to APM */

    if (!(noAPM = APMAttach())) {

        /* if attached, register for suspend and resume */

        APMRegister ((PAPMHANDLER)APMEventHandler,
                     APM_NOTIFYSETPWR | APM_NOTIFYNORMRESUME |
                     APM_NOTIFYCRITRESUME | APM_NOTIFYSTBYRESUME,
                     0);

        /* prepare driver to deal with APM notifications */

    } else {

        /* prepare driver to deal with APM events (like suspend) even if it */
        /* doesn't see any notifications about them! */

    }
}
```





# APM Registration

```
USHORT FAR _cdecl APMEventHandler (PAPMEVENT Event) {
    USHORT Message = (USHORT)Event->ulParm1;
    USHORT PowerState;

    if (Message == APM_SETPWRSTATE) {
        PowerState = (USHORT)(Event->ulParm2 >> 16);
        if (PowerState != APM_PWRSTATEREADY)
            return (APMSuspend (PowerState));
    } else if ((Message == APM_NORMRESUMEEVENT) ||
               (Message == APM_CRITRESUMEEVENT) ||
               (Message == APM_STBYRESUMEEVENT)) {
        PowerState = 0;
        return (APMResume());
    }
    return 0;
}

USHORT NEAR APMSuspend (USHORT PowerState) {
    if (PowerState == APM_PWRSTATESUSPEND) {
        /* prepare hardware and software for suspend */
    }
    return 0;
}

USHORT NEAR APMResume() {
    /* restore/reinitialize hardware and software after resume */
    return 0;
}
```



# PCCard/Cardbus

- The card services subsystem is optional, the driver needs to be prepared not to find it
- the minimum set of card service events to be handled is
  - CLIENTINFO (identify as client driver)
  - CARD\_INSERTION
  - CARD\_REMOVAL
- you may decide to handle more events
- most likely you need an assembler stub



# Card Services Registration

```
/* Attach to PCMCIA.SYS
 * check for presence of card service
 * allocate a context hook for deferred processing of events
 * register driver with card services
 * scan sockets for PCCards already inserted (no card insertion events
   will be generated!)
 */

USHORT NEAR PCMCIASetup() {
    if (SELECTOROF(CSIDC.ProtIDCEntry) != NULL)
        return (FALSE);    /* already initialized */

    if (!DevHelp_AttachDD (PCMCIA_DDName, (NPBYTE)&CSIDC) &&
        CardServicesPresent() &&
        !DevHelp_AllocateCtxHook ((NPFN)&CSHookHandler, (PULONG)&CSCtxHook) &&
        !CSRegisterClient()) {
        int Socket;

        for (Socket = 0; Socket < NumSockets; Socket++)
            if (0 == CSCardPresent (Socket)) {
                PCCardPresent |= (1 << Socket);
            }
        return (FALSE);
    }
    return (TRUE);
}
```



# Card Services Events

```
VOID NEAR _cdecl CScallbackHandler (USHORT Socket, USHORT Event, PCHAR Buffer)
{
    /* release resources if a card removal event occurs */
    /* acquire resources if card insertion event occurs */

    switch (Event) {
        case CARD_REMOVAL:
            if (!InitComplete) return;
            PCCardPresent &= ~(1 << Socket);
            CSUnconfigure (Socket);
            CardRemoval (Socket);
            return;

        case CARD_INSERTION:
            if (!InitComplete) return;
            if (CSConfigure (Socket) == 0) {
                PCCardPresent |= (1 << Socket);
                CardInsertion (Socket);
            }
            return;

        case CLIENTINFO:
            /* fill client info structure */
            return;

        /* handle other events if required */
    }
}
```



# Card Services Events

```
USHORT NEAR CardRemoval (USHORT Socket) {
    /* handle removal of a PCCard
     * - detach hardware from the supporting driver code
     * - release resources allocated to the hardware being removed
     * - prepare driver to handle calls directed at removed hardware *gratiously*
     */
    return (0);
}

USHORT NEAR CardInsertion (USHORT Socket) {
    /* handle insertion of a PCCard (part I)
     * - allocate resources to the hardware being inserted
     * - make a *short* test if hardware is supported and healthy
     * - release resources if test fails
     */

    if (test passed) {
        /* defer full initialization (make take long) */

        DevHelp_ArmCtxHook (Socket, CSCtxHook);

    } else {
        /* release resources */

        CSUnconfigure (Socket);
    }
    return (0);
}
```





# Card Services Events

```
USHORT NEAR _fastcall CardInsertionDeferred (USHORT Socket) {  
  
    /* handle insertion of a PCCard (part II)  
     * - full initialization of the newly inserted PCCard  
     * - release resources if initialization fails  
     */  
  
    if (initialized) {  
  
        /* attach hardware to the supporting driver code */  
  
    } else {  
  
        /* release resources */  
  
        CSUnconfigure (Socket);  
    }  
    return (0);  
}
```



# Card Services Stub

```
; VOID FAR _cdecl CScallbackStub()  
; VOID NEAR _cdecl CScallbackHandler (USHORT Socket, USHORT Event,  
;                                     PCHAR Buffer)  
  
PUBLIC _CScallbackStub  
  
_CScallbackStub PROC FAR  
  
    PUSHF  
    PUSHA  
    PUSH    DS  
    PUSH    ES           ; setup buffer pointer  
    PUSH    BX  
    AND     AX, 00FFh  
    PUSH    AX           ; set up event number,  
    PUSH    CX           ; socket number  
    MOV     AX, _DATA    ; and data segment  
    MOV     DS, AX  
    CALL    _CScallbackHandler  
    ADD     SP, 3*2  
  
    POP     ES  
    POP     DS  
    POPA  
    POPF  
    RET  
  
_CScallbackStub ENDP
```



# Resource Management

- From a device driver's view, resource manager offers two basic services:
  - maintaining and validating hardware resources
  - maintaining and looking up the device database
- the former is mandatory, the latter is optional



# Resource Management

- As a bare minimum, your driver must register with Resource Manager if it stays resident
- for users, this is the only way to find out if a driver is actually loaded by means of tools provided by a standard OS/2 installation (i.e. RMView /D)
- for developers, this is the easiest way to find out which hardware resources a driver is operating on



# Resource Management

## Resources

- every device driver creates a driver object in RM. Software only drivers are done here.
- for each device, drivers allocate hardware resource objects in RM and check for collisions
- in case of success, drivers:
  - create an adapter object for each hardware instance they handle and assign them to the driver object
  - assign hardware resources to the adapter objects
  - possibly create and assign device objects





## Resource Management Hardware Look-up

- Device drivers may look up the OS/2 device database for supported hardware
- this database is updated at each boot or on demand by means of the snoopers drivers (SNOOP.LST)
- the hardware look-up may be for **exact** matches of device identifiers (PnP, PCI, EISA) or for **compatible** devices (f.e. »looks like an IDE port«)



# OEMHelp Services

- OEMHelp provides access to configuration type BIOS services:
  - query video info
  - query MCA and ESCD info
  - enumerate and configure PCI adapters
- the OEMHelp PCI functions are more appropriate for device discovery in case of class specific drivers or info not maintained by the snoopers



# OEMHelp Services

- OEMHelp may be called at DEVICE init time (ring 3) through regular 16-bit DosCalls
- OEMHelp needs to be called at BASEDEV init or task time (ring 0) through an IDC interface. This requires an assembler stub.



# OEMHelp Services

```
CHAR      OEMHLP_DDName[9] = "OEMHLP$ ";
IDCTABLE OemHlpIDC        = { 0 };

UCHAR SetupOEMHlp() {
    if ((SELECTOROF(OemHlpIDC.ProtIDCEnt) != NULL) &&
        (OemHlpIDC.ProtIDC_DS != NULL))
        return (0);    /* already initialized */

    /* Setup Global OEMHlp Variables */
    if (DevHelp_AttachDD (OEMHLP_DDName, (NPBYTE)&OemHlpIDC))
        return (1);    /* Couldn't find OEMHLP's IDC */

    if ((SELECTOROF(OemHlpIDC.ProtIDCEnt) == NULL) ||
        (OemHlpIDC.ProtIDC_DS == NULL))
        return (1);    /* Bad Entry Point or Data Segment */
    return (0);
}

/* example */
{
    RP_GENIOCTL IOCtrlRP;

    /* Setup IO Control Packet here */

    return (CalloEMHlp ((PRPH)&IOCtrlRP));
}
```



# OEMHelp Services

```
; USHORT FAR _fastcall CalloOEMHlp (PRPH pRPH);
```

```
@CalloOEMHlp PROC FAR
```

```
    PUSH    BP
    MOV     BP, SP
    PUSH    SI
    PUSH    DI
    LES     BX, DWORD PTR [BP+6]
    TEST    WORD PTR [_OemHlpIDC.Entry], -1
    JNZ     DoOEMHlp
```

```
    MOV     AX, 8100h
    MOV     WORD PTR ES:[BX+3], AX
    JMP     SHORT CalloOEMHlpEnd
```

```
DoOEMHlp:
```

```
    PUSH    [_OemHlpIDC.Entry]
    PUSH    DS
    MOV     DS, [_OemHlpIDC.DSeg]
    CALL    DWORD PTR [BP-8]
    POP     DS
    ADD     SP, 4
    MOV     AX, WORD PTR ES:[BX+3]
```

```
CalloOEMHlpEnd:
```

```
    AND     AX, 8000h
    POP     DI
    POP     SI
    LEAVE
    RET     4
```

```
@CalloOEMHlp ENDP
```